YOUR COMPUTER IS NOW STONED (...AGAIN!). THE RISE OF MBR ROOTKITS

Kimmo Kasslin
F-Secure Security Labs, Suite 2A-5-2, Level 5, Block 2A, Plaza Sentral, Jalan Stesen Sentral 5, 50470 Kuala Lumpur, Malaysia

Elia Florio
Symantec Security Response, Ballycoolin Business Park, Blanchardstown, Dublin 15, Ireland

Email kimmo.kasslin@f-secure.com, elia_florio@symantec.com

ABSTRACT

The war against invisible malware has been taken down to a new battleground, the lowest level seen so far in the wild: the Master Boot Record. The MBR rootkit, a.k.a. Mebroot, appeared in the wild in December 2007 and rapidly evolved from earlier beta versions to a fully working malware product. The Mebroot rootkit uses techniques never before seen in modern threats and so it can be considered the next generation of stealth rootkit and kernel infector, written by professional malware developers and clearly not for fun. The most notable characteristic of Mebroot is the fact that it replaces the system's Master Boot Record with malicious code that owns the machine completely from the boot, before the operating system itself gets loaded. Years after Stoned and Michelangelo, Master Boot Record infection has been reborn with Mebroot on modern platforms. However, this technique is only the tip of the iceberg of a bigger cybercriminal project, since the final goal of Mebroot is to download and install additional banking trojan components on the infected machine. In this paper we present an extended view of the MBR rootkit's features and its evolution – including a detailed look at its disk stealth, firewall bypassing, anti-analysis and anti-detection techniques.

MEBROOT EVOLUTION

We can trace back the first evidence of 'Mebroot' (the MBR rootkit) to the end of 2007. According to the PE timestamp of the oldest sample seen, it was compiled in the early days of November 2007 and distributed multiple times before the end of the year. An interesting timeline of Mebroot evolution was first outlined by Matt Richard from *iDefense* [1], who initially discovered the first sample in the wild together with the *GMER* team [2].

What we know is that during November 2007 a well-known malicious domain (hxxp://gfeptwe.com) that had been used in the past to distribute and install variants of Trojan.Anserin (a.k.a. Sinowal or Torpig) began to serve copies of the MBR rootkit for a limited period of time. The malware was installed via drive-by exploits using a set of old *Microsoft* vulnerabilities, probably to stay under the radar during this 'beta' release stage. The whole timeline reads like a big development and malware QA plan; in fact all the samples released in the initial period have close PE timestamps and very small changes in the code. Two waves of drive-by attacks related to Mebroot took place between December 2007 (Mebroot 'v.0' or beta release) and January 2008

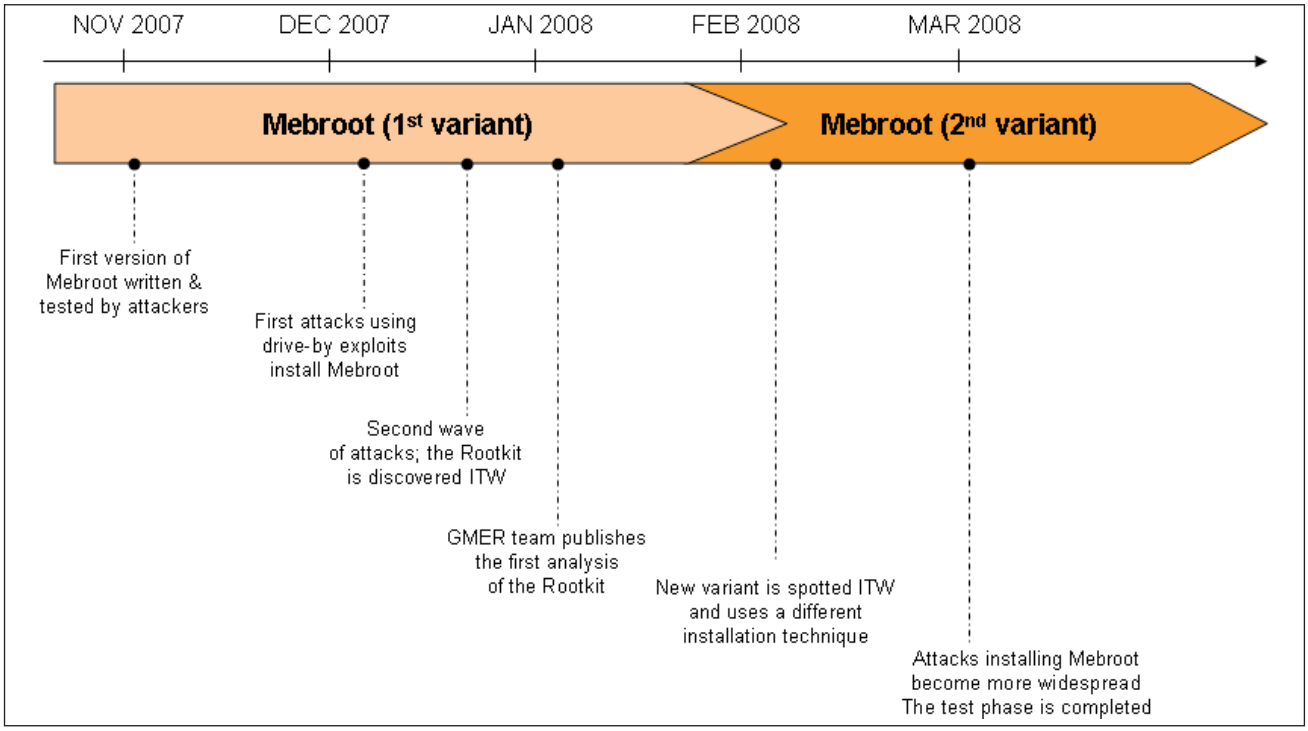

Figure 1: Timeline of Mebroot evolution from 'beta' to final release.

(Mebroot 'v.1' release). These attacks were followed by a period of calm, probably as a result of the popularity and media coverage gained by the rootkit in January. In March 2008 [3] the flow of attacks installing Mebroot resumed steadily, this time spreading a second, improved variant ('v.2' release) of Mebroot which uses aggressive countermeasures against all AV tools and anti-rootkit programs created to detect and defeat earlier variants. In June 2008 during our analysis we also found some improvements and additions to the rootkit code that led us to believe that we were in the middle of a 'v.3' release.

PACKER CHANGES OVER TIME

AV researchers and the entire security community have clearly been impressed by the efforts of Mebroot's development team. The gang has been very active and worked with care on each aspect of the 'final product'. This fact makes Mebroot different from other simple backdoor programs in the wild. The developers constantly improved the rootkit code with new features, new stealth tricks, anti anti-rootkit routines and even code optimization and memory checks to avoid blue-screen errors.

At the same time they worked to enhance the dropper/installer routine with a new technique (discussed in the following sections of this paper) and also modified the polymorphic packer used to encrypt EXE installers. During its entire life cycle, the polymorphic packer evolved from simple spaghetti JMP instructions, to complicated polymorphic artefacts mixed with anti-emulation tricks based on exceptions and fake API calls. This nasty packer scrambles all the execution flow of a program by interleaving valid opcodes with JMP or JMP DWORD[addr] instructions. The final effect is to have a piece of polymorphic code which is very difficult to trace and analyse, but with the same functionality.

Analysis of the packer reveals a code that allocates one large and one small memory buffer using VirtualAlloc(). The packer puts a large sequence of bytes into the small buffer using a pseudo-random generator algorithm with an initial seed key. This buffer is used to perform the decryption of a specific file section which gets decoded into the second allocated buffer to form a proper executable file. The executable image is relocated in memory with valid imports and is executed from an entry point just after the final VirtualFree() call of the packer. This crazy 'spaghetti-like' packer has already been used by the Trojan.Anserin malware family and is used to protect against reversing both EXE and SYS samples (yes, the gang created a kernel-mode version of this packer too).

OWNING THE MBR: HOW IT STARTED

It is unclear how long it has taken the authors to develop and write the code of this sophisticated threat, but the idea of malicious code that modifies a system's MBR is not brand new and was discussed some years ago. In 2004 Greg Hoglund wrote about MBR attacks in his book *Exploiting Software* [4], while the most notable research in the area of MBR rootkits was undertaken by Derek Soeder of *eEye* [5] during 2005. Soeder created BootRoot, a proof-of-concept MBR rootkit able to target *Windows XP* and *2000*. Finally, researchers Nitin and Vipin Kumar of *NVLabs* recently published a paper [6] about a new type of MBR rootkit called Vbootkit, designed expressly to work on *Windows Vista*. It is quite obvious that Mebroot's authors benefited from all these previous pieces of research and this fact is confirmed by a quick comparison of the MBR code

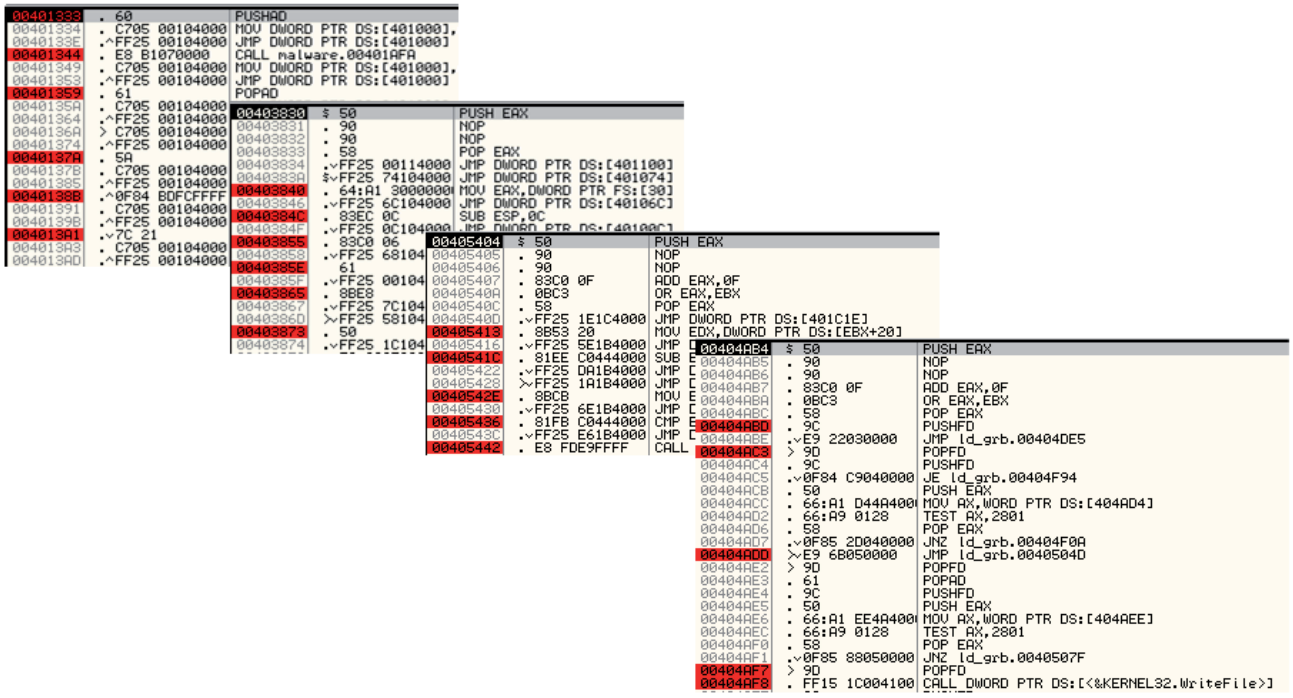

Figure 2: Evolution of the Mebroot packer over time.

of Trojan.Mebroot and BootRoot. A large area of the MBR loader is almost identical to the BootRoot code published by *eEye*. Mebroot's MBR code hooks INT 13 at boot exactly as BootRoot does with the intent of patching the OSLOADER image (part of the NTLDR file) when it gets loaded. This patch is done on the fly with the same static signature used by BootRoot (8BF085F6742?803D). The signature is patched with a CALL DWORD[addr] instruction that gives control to the second stage payload of the malware. This payload will be discussed in more detail in the next sections.

Raw disk access under Windows

Mebroot arrives with an EXE installer that is typically between 250 KB and 350 KB for earlier variants and up to 430 KB for recent variants. It takes control of the system by overwriting the MBR. This attack is possible because the Master Boot Record is still a weak point of modern OS architectures. Mebroot variant 'v.0' initially used a standard and documented way to read/write MBR and raw disk sectors with normal *Windows* APIs. However, Mebroot variants 'v.1' released after February 2008 moved to a new and very sophisticated installation procedure which may bypass HIPS programs and try to stay under the radar. Both techniques are described in detail in the following paragraphs.

THE EASY WAY: OWNING A DISK WITH CREATEFILE()

Some versions of *Windows* let programs overwrite disk sectors (including the MBR) directly and without proper restrictions. The initial reports about this MBR attack were a bit confused, so let's clarify some facts to understand when the attack is possible. On *Windows 2000*, *XP* and *2003* systems, raw access to disk is possible for any user-mode program running in ring-3 (no need to go in ring-0!), but this requires Administrator [7] privileges! The fact that most users run *Windows* as Administrator makes them clearly vulnerable to this type of rootkit.

The issue has been known about for some time for the *2K*/*XP* families, while *Vista* was partially secured in 2006 (with Release Candidate 2) after a successful attack demonstration made by Joanna Rutkowska, known as the Pagefile Attack [8]. In fact, the attack is now mitigated on *Vista* by UAC, which blocks raw access to disks. Table 1 summarizes which operating systems can be infected by Mebroot.

Windows OS	Can MBR be infected?	Is rootkit active?
Windows 2000 (user is Administrator)	YES	YES
Windows XP (user is Administrator)	YES	YES
Windows 2003 (user is Administrator)	YES	YES
Windows Vista (UAC disabled)	YES	NO
Windows Vista (UAC enabled)	NO	NO

Table1: Windows versions and MBR rootkit.

It is important to clarify that Mebroot can infect the *Vista* MBR only if UAC is disabled, however the rootkit, even after a successful infection, will not be able to load itself at boot because it targets specific signatures of the *Windows* kernel not present on *Vista*. In this scenario *Vista* users may live with an infected MBR that boots up the operating system normally without seeing any rootkit activity, because the malware would never be loaded in memory. In addition to this, *Vista* is also secure because its boot process is completely different from any previous OS. It is possible that future variants of this threat may overcome this limitation.

THE HARD WAY: DISK.SYS WRAPPER AND THE SETWINEVENTHOOK() TRICK

The latest variant of Mebroot spotted after February 2008 uses a different approach to perform raw operations on disk. Instead of using CreateFile(), Mebroot loads a driver that works as a 'wrapper' for the system I/O driver disk.sys. Essentially the new installer uses a kernel driver to communicate with the real OS disk driver and to perform low-level read/write using IRP communications.

This change in the installation technique is probably motivated by the fact that HIPS and active protection systems started to block suspicious CreateFile() operations on physical hard drives. Using a kernel driver as a 'wrapper' to read/write the disk can overcome this limitation, but many researchers may notice an important fact: how can Mebroot load this 'wrapper' driver on the system? Is it really a good strategy? Loading a kernel driver may be flagged as suspicious as well and so is pointless. The complete loading and infection strategy is described step by step with the help of event-driven diagrams in Figures 3 and 4.

The Mebroot installer file ('ld.exe' in this example) first takes care of scheduling a self-delete operation at reboot using MoveFileEx() and then drops an executable ('1.tmp') in the %TEMP% folder and runs it. This second program is the user-mode MBR infector component and runs a waiting loop which keeps trying to create a non-existent device named \\.\RealHardDisk (N.B. the real symbolic link to the hard-disk is \\.\PhysicalDrive). While '1.tmp' keeps trying and re-trying without success, the installer 'ld.exe' runs itself with the parameters '--cp 2.tmp'. The effect of this parameter is to force 'ld.exe' to copy itself as '2.tmp' with the DLL bit set. Essentially the file '2.tmp' is a DLL version of 'ld.exe'. At this stage of installation 'ld.exe' continues execution by reading USER32.DLL from disk and checking if the first five bytes of SetWinEventHook() match between memory and disk images. If not, it restores the original five bytes to unhook the API (typically hooked by HIPS and security programs). This particular API is an unusual and not well-documented trick to inject a DLL into the *Explorer* process context without active injection (e.g. OpenProcess/CreateRemoteThread). The installer calls this API passing three important parameters: *Explorer* process id, the path of the '2.tmp' DLL file and the export routine 'wep()'.

After that, the Mebroot installer code runs completely from the *Explorer* process space and so any changes to the OS

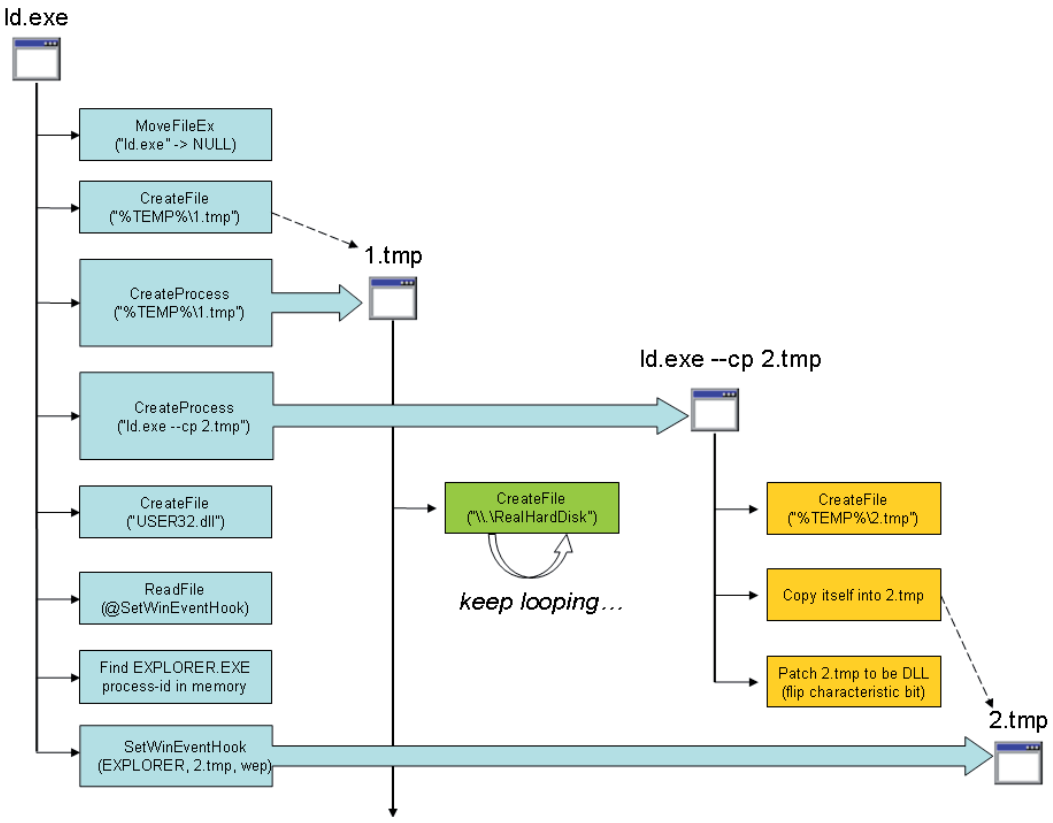

Figure 3: Mebroot installer technique with SetWinEventHook().

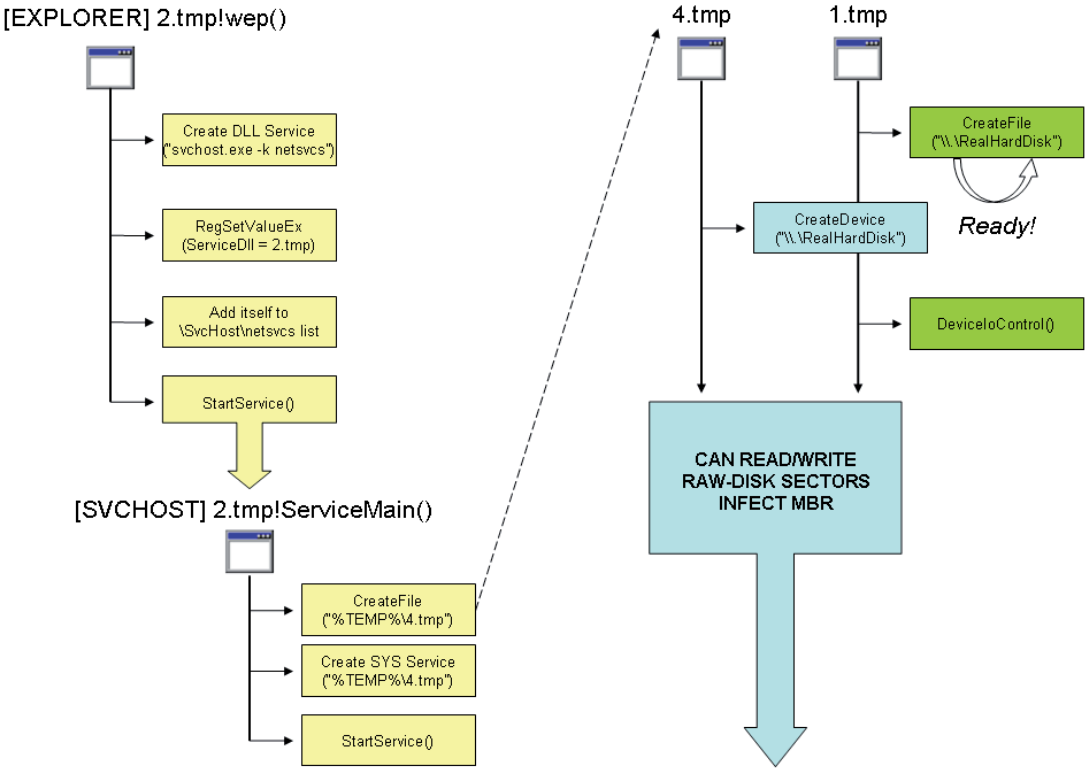

Figure 4: Mebroot installer code injected into EXPLORER and SVCHOST.

```
0040172B 68 00 00 07 00                    push    IOCTL_DISK_GET_DRIVE_GEOMETRY ; dwIoControlCode
00401730 FF 75 08                          push    [ebp+hPhysicalDriveX] ; hDevice
00401733 FF 15 20 10 40 00                 call    ds:DeviceIoControl
00401733
00401739 85 C0                             test    eax, eax
0040173B 0F 84 F9 03 00 00                 jz      exit_EAX_0
0040173B
00401741
00401741                           checkHD_BytesPerSector:
00401741 BB 00 02 00 00                    mov     ebx, 512
00401746 39 5D D8                          cmp     [ebp+outbuf_DiskGeom.BytesPerSector], ebx
00401749 0F 85 EB 03 00 00                 jnz     exit_EAX_0
00401749
0040174F 56                                push    esi             ; lpOverlapped
00401750 8D 45 F8                          lea     eax, [ebp+NumberOfBytesRead]
00401753 50                                push    eax             ; lpNumberOfBytesRead
00401754 53                                push    ebx             ; 512
00401755 8D 85 C4 FD FF FF                 lea     eax, [ebp+buf_OriginalMBR]
0040175B 50                                push    eax             ; lpBuffer
0040175C FF 75 08                          push    [ebp+hPhysicalDriveX] ; hFile
0040175F FF 15 18 10 40 00                 call    ds:ReadFile
0040175F
00401765 85 C0                             test    eax, eax
00401767 0F 84 CD 03 00 00                 jz      exit_EAX_0
00401767
0040176D 39 5D F8                          cmp     [ebp+NumberOfBytesRead], ebx
00401770 0F 85 C4 03 00 00                 jnz     exit_EAX_0
00401770
00401776
00401776                           check_BootMagic:
00401776 66 81 7D C2 55 AA                 cmp     [ebp+buf_OriginalMBR.BootMagicSignature], 0AA55h
0040177C 0F 85 B8 03 00 00                 jnz     exit_EAX_0
```

Figure 5: Raw access to disk with CreateFile() API.

configuration do not appear suspicious because they are made by *Explorer* itself. The function 'wep()' performs another step in the whole infection strategy by creating a new DLL service named '{BEE686B9-4C84-4487-9D72-9F40F051E973}' which is added to the list 'HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost\netsvcs'. The service is started immediately and this will execute ServiceMain() of the '2.tmp' module into SVCHOST context with full OS privileges enabled. The Mebroot installer is now ready to drop the 'wrapper' driver (4.tmp) and register it with the name '{DEF85C80-216A-43AB-AF70-1665EDBE2780}' as a kernel-mode service. When the kernel driver runs, it creates the device \\.\RealHardDisk and is immediately synchronized with '1.tmp' which was waiting for exactly this moment to start the infection. Read and write operations to disks are performed using DeviceIoControl() requests to the 'wrapper' driver. At the end of the infection, the installer 'ld.exe' takes care of deleting all the files and services created, to avoid suspicious traces.

OWNING THE MBR

Mebroot tries to infect the first 16 disk drives connected to the machine. The side effect of this behaviour is that in some cases the rootkit also infects external USB disks and hard drives (so can be considered a worm?). Infected external disks would not have an active infection, because typically they are not used to boot the operating system, but the disk will still contain traces of the malware on some sectors.

During the installation phase, the malware first reads the current disk MBR and checks some characteristics of the drive like the number of bytes per sector (it expects 512 bytes), the magic boot signature 0x55AA at the end of the MBR and whether the drive has already been infected (the infection marker is the DWORD 0xAD022C83 at offset 0x16 of the MBR). Next, it parses the partition table to find the physical end of the disk and it verifies that the un-partitioned slack space at the end is sufficient to write its own malicious code. The installer usually needs at least ~650 free sectors that will be used to store the main rootkit driver. This strategy is clever for two reasons: the driver is not stored as a file on the system, but it is stored in raw disk sectors. Secondly, writing the malicious driver after the end of the disk means that some forensic expertise is required to extract samples from infected machines. The installer makes a note of the sector where the rootkit executable is stored and then adjusts in memory its Payload Loader shellcode that will load the SYS driver at the next reboot. Finally, it overwrites three sectors immediately before the beginning of the first partition. On *Windows 2000* and *XP* with a single partition, Mebroot typically overwrites sectors 60, 61 and 62. These sectors could be different on systems with multiple OS and disk partitions.

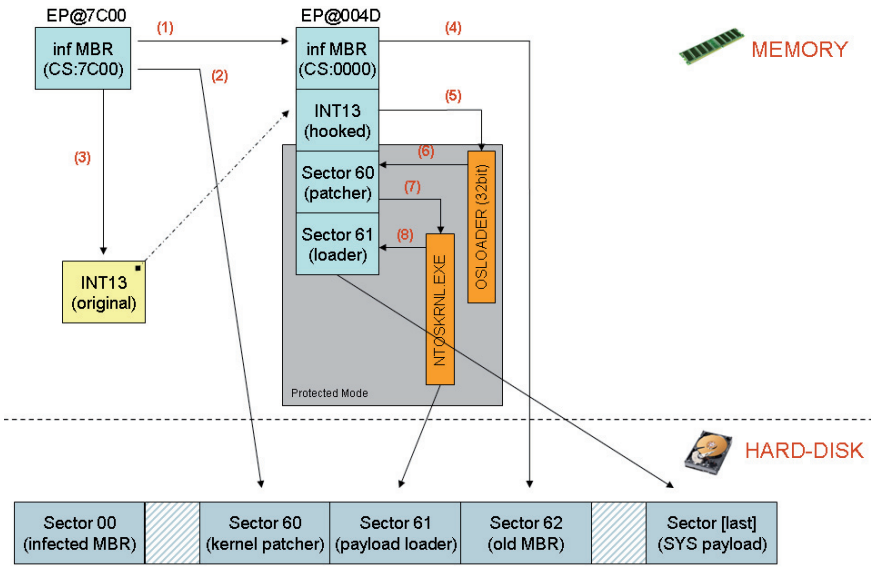

Figure 6: Mebroot loading process – how to own the system from the boot.

OWNING THE SYSTEM FROM THE BOOT

The complete scheme of the Mebroot loading process is shown in Figure 6.

A step-by-step description of the rootkit boot process and kernel infection follows:

1. The infected MBR reserves 2 KB of conventional memory and relocates itself from 0x7C00 to 0x0000.

2. Next, it reads payloads from sector 60s (kernel patcher) and 61 (payload loader) into memory blocks adjacent to the relocated code.
3. The MBR code hooks INT 13 and passes control to relocated code at 0x004D address.
4. It reads sector 62 (old MBR) back to 0x7C00 memory and passes control to it; the OS starts booting up normally while INT13 is hooked by the threat.
5. The hooked code intercepts all disk-reading operations and patches the OSLOADER module (part of NTLDR) when it gets loaded from disk.
6. The patched OSLOADER calls the Kernel Patcher shellcode in memory (sector 60).
7. This shellcode scans and patches the NTOSKRNL.EXE image near 'CALL nt!IoInitSystem'.
8. The modified NTOSKRNL.EXE calls the Payload Loader shellcode (sector 61) which loads and runs the rootkit driver stored in the last sector of the disk.

To minimize footprints and traces in memory, the loader shellcode takes care of deleting itself by filling up with zeroes the memory area where it is stored. This detail leads us to believe that nothing is left to chance and the authors of this nasty piece of code are skilled and meticulous malware programmers.

Analysing the final rootkit driver loaded in memory requires some extra effort. Some rootkit variants in fact have an extra packing layer that unpacks the real kernel driver using scrambled spaghetti code. Also in this case, a good breakpoint on ExAllocatePoolWithTag will do the job and let us dump the final unpacked driver. Since the rootkit SYS driver is loaded by its own loader in an unusual way, the module does not expect the normal parameters passed to *Windows* drivers. In fact, it receives three parameters passed by the Payload Loader: the kernel ImageBase of the unpacked driver, a pointer to PsLoadedModuleList (used to resolve imports) and the ImageBase of the packed driver. The rootkit takes care of resolving all NTOSKRNL and HAL imports with its own routine and also deletes from memory the packed driver image when it is no longer needed. Later on, even the MZ header of the unpacked driver is deleted from memory to minimize footprints, leaving in the kernel space only random traces of code in executable memory pages.

DISK STEALTH TECHNIQUE OF THE FIRST VERSION

The first variant of Mebroot hides itself by hooking the disk.sys driver. It finds DeviceObject for \Device\HardDisk[N]\DR0 and reads the old MBR from Sector 62 into an allocated pool that will be used as 'cached copy' of the old MBR to improve the performance of stealth operations. Since the rootkit does not have files, process or registry keys to hide, the stealth functionalities are limited to intercepting read/write operations on raw disk sectors. This is done by hooking the dispatch handlers of \Driver\Disk for IRP_MJ_READ and IRP_MJ_WRITE routines. When a program tries to read the MBR (sector 00) or any other sector used by the rootkit (60, 61, 62 or sectors after the end of the disk) the hooked code will return a fake image of the sector, showing the old MBR or eventually an empty sector filled up with zeroes for the other cases. In a similar way, the rootkit will protect itself by blocking all write operations to its sectors. The rootkit needs to maintain a hook-free version of the IRP_MJ_READ and IRP_MJ_WRITE functions and so it uses a special trick: it generates a random DWORD value used as a 'magic key'. Later, the rootkit is able to perform normal read/write operations with the original dispatch routines simply by calling the disk.sys driver with an IRP packet that contains this magic key at offset 0x40.

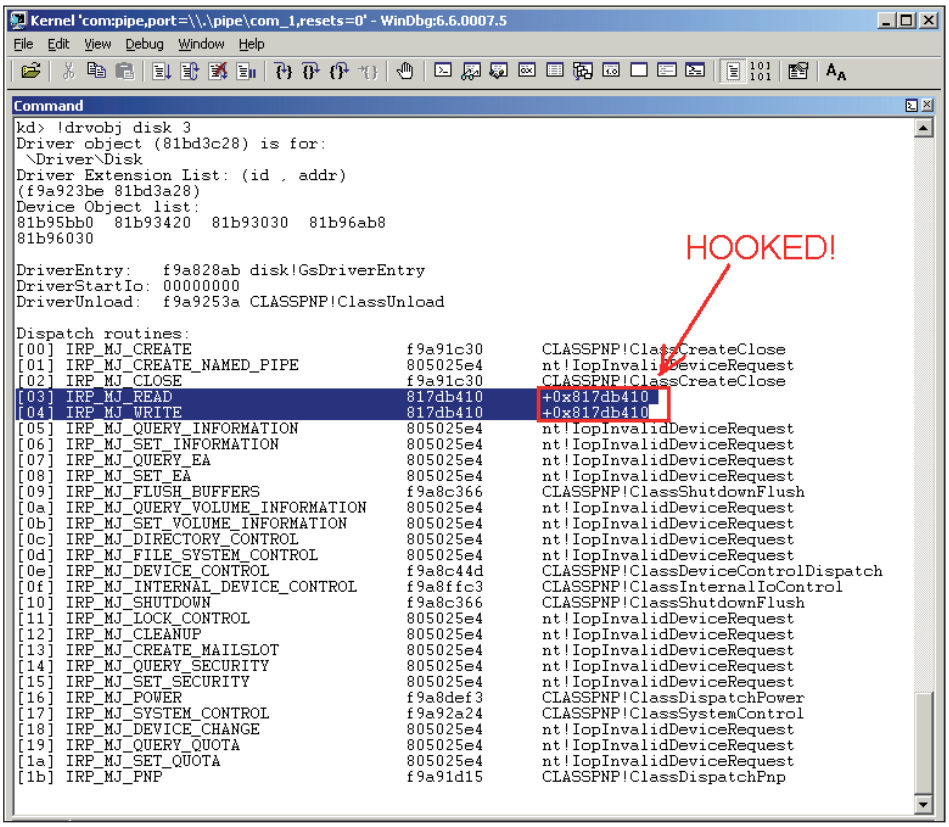

Figure 7: Two hooked pointers in the DISK.SYS dispatch table are suspicious enough.

STEALTH TECHNIQUE ENHANCEMENTS

In March 2008 the response from AV companies to this new

challenge was already available to users. AV programs and free stand-alone tools were able to detect and remove active Mebroot infections from computers. The criminal master plan was obviously compromised and so the Mebroot authors decided to take a further step and improved the stealth abilities of their creature. Many of the new features that were introduced in the second rootkit variant are discussed in [9].

The first variant of Mebroot targeted only IRP_MJ_READ and IRP_MJ_WRITE function pointers in the MajorFunction table of DISK.SYS. The weaknesses of this hooking technique were immediately visible to AV researchers and different approaches to detect compromised kernels were introduced:

(A) Anomalies between addresses and module ranges of IRP_MJ_* pointers in the DISK.SYS MajorFunction table (see Figure 7).

(B) Discrepancies between the IRP_MJ_READ/IRP_MJ_WRITE pointers used by DISK.SYS and by CDROM.SYS.

(C) Discrepancies between the IRP_MJ_READ/IRP_MJ_WRITE pointers used by DISK.SYS and values initialized by the CLASSPNP.SYS routine ClassInitialize().

Mebroot's authors reversed the code of many removal and detection programs and in March 2008 they released into the wild a variant that was able to overcome (A), (B) and (C). The irregularities in (A) have been fixed by the authors by adding dummy pointers for all IRP_MJ_* functions of DISK.SYS. Instead of hooking only two pointers, the rootkit now hooks all the pointers in the MajorFunction table. The pointers of IRP_MJ_READ/WRITE are redirected to the rootkit routine handler, while all the other pointers (still in the same module) are hooked with a dummy handler which jumps back to the original DISK.SYS handler for each of them. A similar approach was used to fix the discrepancies in (B). In fact, the rootkit now hooks the CDROM.SYS dispatch table with dummy pointers with a replica of the DISK.SYS pointers. Finally, to avoid restoration of the original pointers from the CLASSPNP.SYS driver, the gang added an extra routine (shown in Figure 8) which scans code sections of the CLASSPNP driver in memory and patches the original pointers of the ClassInitialize() routine with rootkit hooked pointers.

In the middle of March we found an improved variant that introduced a 'watchdog' thread active in memory. Any attempt by AV or removal tools to restore the original read/write IRP pointers is monitored by the watchdog thread which immediately restores the rootkit pointers and reinfects the MBR and disk sectors. This implementation has some known bugs explained in [9] but it was effective enough to make removal a complicated job. In later variants some of these known bugs have been fixed.

At the beginning of June we found a new Mebroot variant that modifies the watchdog thread's _ETHREAD->StartAddress structure member and makes it point to an existing benign system thread start address. This was probably done to prevent AV or removal tools from being able to find the watchdog thread by enumerating system threads and looking to see whether their start addresses pointed outside of trusted module address space.

Other attempts to evade detection in the MBR were also found in the MBR loader code which may contain a random amount of NOP opcodes interleaved between instructions to evade static string signatures.

FIREWALL-BYPASSING TECHNIQUES

Analysing the rootkit driver's network code is one step harder. The majority of its functions are still heavily obfuscated, even after successful unpacking. The fastest way to get past the obfuscation is code tracing and custom scripts to clean up the trace logs of extra garbage. After lots of frustrating moments and some breakthroughs we now know that Mebroot's firewall-bypassing technique is taken one step further than that which we described in our article about Srizbi [10].

Mebroot also operates in the NDIS layer but it uses a different approach to gain

```
loc_161A3:                              ; DATA XREF: sub_3DBB0+CEE↓o
                lea     eax, [ebp+ModuleSize]
                push    eax             ; outModSize
                lea     eax, [ebp+ModuleImgBase]
                push    eax             ; outModBase
                push    offset szClasspnp_sys ; szModuleName
                call    FindModuleByNameNtQuerySystemInfo
                xor     edi, edi
                cmp     eax, edi
                jl      exit_retn
                lea     eax, [ebp+pPE_SectHdr]
                push    eax
                push    edi             ; NULL
                lea     eax, [ebp+pPE_FileHdr]
                push    eax
                push    [ebp+ModuleSize]
                push    [ebp+ModuleImgBase]
                call    ValidatePEImgAndGetHeaders ;
                                        ; ([in]Base, [in]Size,
                                        ;  [out]PE_FileHdr, [out]PE_OptHdr, [out]PE_SecHdr)
                                        ;
                test    al, al
                jnz     short classpnp_module_found
                mov     eax, STATUS_UNSUCCESSFUL
                jmp     short exit_retn
; ---------------------------------------------------------------------------

classpnp_module_found:                  ; CODE XREF: HookClassPNP+39↑j
                mov     eax, [ebp+pPE_FileHdr]
                cmp     [eax+IMAGE_FILE_HEADER.NumberOfSections], di
C0              mov     [ebp+retn_value], STATUS_UNSUCCESSFUL
                mov     [ebp+counter_section], edi
```

Figure 8: Mebroot's dedicated patch routine for CLASSPNP.SYS pointers.

access to the internal NDIS structures. Whereas Srizbi installed a dummy protocol, Mebroot uses the unexported ndisMiniDriverList global which points to the head of a linked list consisting of installed minidriver objects. These objects are described by the _NDIS_M_DRIVER_BLOCK structure which is shown below:

```
NDIS!_NDIS_M_DRIVER_BLOCK
  +0x000 NextDriver    : 0x81b04290 _NDIS_M_DRIVER_BLOCK
  +0x004 MiniportQueue : 0x81a61828 _NDIS_MINIPORT_BLOCK
  …
```

The MiniportQueue structure member points to miniport block objects bound to the specific driver. By traversing through the ndisMiniDriverList linked list Mebroot has access to all miniports and from there onwards it uses a similar approach to Srizbi to find a suitable adapter that is bound to either the PSCHED or TCP/IP protocol. Before Mebroot accesses the linked list it gets exclusive access to it by acquiring the ndisMiniDriverListLock spinlock which is also unexported. This is a good example of the level of professionalism the Mebroot authors are practising since accessing any linked list without exclusive access will pose a risk of system crash if any of the link pointers are modified simultaneously.

Finally, after Mebroot has found a suitable protocol to hijack, it finds the address of the lowest level send handler function and hooks three NDIS handler functions.

To send packets it uses the following handler function:

```
NDIS!_NDIS_M_DRIVER_BLOCK
  +0x020 MiniportCharacteristics : _NDIS51_MINIPORT_
CHARACTERISTICS
  +0x040 SendPacketsHandler : 0xf9adf332 void pcntpci5
!LanceSendPackets+0
```

To get a notification after the send operation has completed it uses the following hook:

```
NDIS!_NDIS_MINIPORT_BLOCK
  +0x0ec SendCompleteHandler : 0x81825bb0 void mbr_
rootkit!Hook_SndCompHdlr
```

To receive packets it uses the following hooks:

```
NDIS!_NDIS_OPEN_BLOCK
  +0x040 ReceiveHandler : 0x8182cd10  int  mbr_
rootkit!Hook_RcvHdlr
  +0x050 ReceivePacketHandler : 0x8182e400  int  mbr_
rootkit!Hook_RcvPcktHdlr
```

Mebroot's network code is advanced in many ways. It is powerful – we are not aware of a single firewall product that intercepts calls at the minidriver level which Mebroot uses to send outbound packets. Therefore it is no surprise that all personal firewall products we were able to test were fully bypassed. It is stealthy – only a single pointer is hooked at all times. The rest of the hooks in the selected protocol's _NDIS_OPEN_BLOCK structure are only in use when the rootkit is sending packets. It accomplishes this by creating a copy of the original open block structure which is then hooked. When it needs to send a packet it replaces a single pointer from the _X_BINDING_INFO structure to point to its private open block structure to make sure the received packets from that point onwards will be processed by its own handler functions. After the packets have been processed the original pointer is put back. This process is illustrated in Figure 9.

Another example of Mebroot's stealth is the way it ensures that all the NDIS API functions it relies on are not hooked by firewalls. Instead of just copying the original ndis.sys from disk into allocated memory and using it as its private module as Srizbi did it uses a 'code pullout' technique to load only the relevant parts of the code into memory. This technique was first described by Alexander Tereshkin, a.k.a. 90210 at rootkit.com [11]. After all relevant code blocks are copied into one continuous block of memory Mebroot relinks all relative call and branch instructions, fixes all relocations to point to the original NDIS module and finally patches its own import address table to make sure all imported NDIS API functions point to the code that was pulled out. This makes runtime analysis and forensics more challenging as it is difficult to locate the relevant NDIS code used by Mebroot since it is just a bunch of bytes somewhere in an allocated non-paged pool.

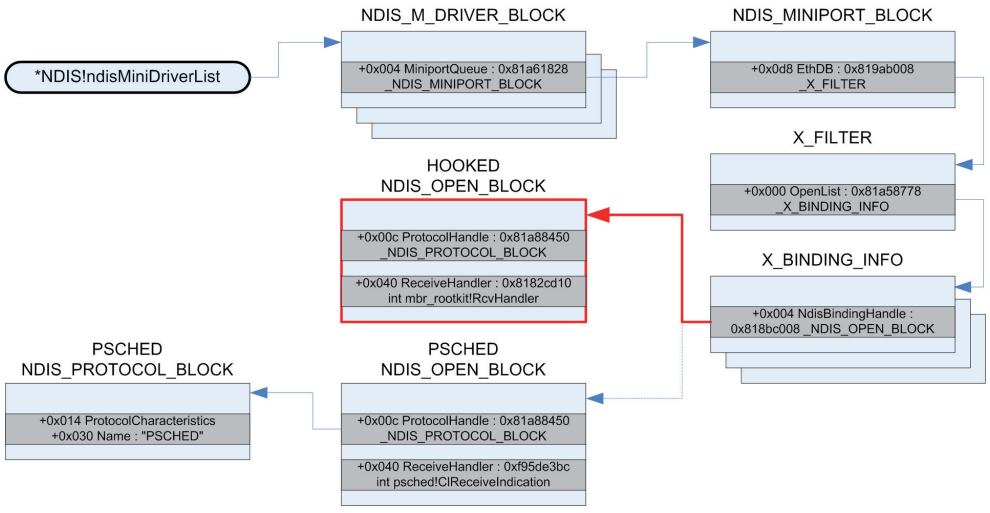

Figure 9: Mebroot activates the full set of its hooks only when it needs them.

ACCESSING UNEXPORTED SYMBOLS

In the previous section we described briefly how Mebroot uses two unexported globals from the NDIS module, namely NdisMiniDriverList and NdisMiniDriverListLock, to get access to all required low-level NDIS structures. Since these symbols are not exported it poses a challenge to the code to find their correct memory addresses in a generic way that is not dependent on the operating system version. The most

common approach so far used by malware has been to scan the module for a specific signature to locate the part of the code where the symbol is used. For example, Rustock scanned 0x1000 bytes from NDIS! NdisMRegisterMiniport onwards and looked for a hardcoded four-byte signature to find the addresses of NdisMiniDriverList and NdisMiniDriverListLock. This approach worked most of the time but in our tests we did encounter blue screens on some systems depending on their batch level which were caused by the malware code when it passed an invalid spinlock pointer to the KfAcquireSpinLock call. As a side note – Rustock was the first malware to utilize this list for enumerating installed minidrivers and miniport blocks bound to them.

```
PAGENPNP:0001D55A    call    @ndisReferencePackage@4 ; ndisReferencePackage(x)
PAGENPNP:0001D55F    mov     esi, offset _ndisMiniDriverListLock
PAGENPNP:0001D564    mov     ecx, esi
PAGENPNP:0001D566    call    ds:__imp_@KfAcquireSpinLock@4 ; KfAcquireSpinLock(x)
PAGENPNP:0001D56C    mov     ecx, _ndisMiniDriverList
PAGENPNP:0001D572    mov     bl, al
```

Figure 10: Both unexported globals are used by NdisRegisterMiniportDriver which is called by the exported NdisMRegisterMiniport function.

To overcome this obvious limitation and to minimize the risk of system crashes Mebroot uses a more sophisticated technique to find the addresses for these two unexported globals. Instead of using simple byte signatures to locate the right place Mebroot analyses the code structure of the exported NdisMRegisterMiniport function which has references to both NdisMiniDriverListLock and NdisMiniDriverList as we can see from Figure 10.

Once more, Mebroot's authors have reused existing code from Tereshkin's PHIDE2 proof-of-concept program. We were able to find clear evidence that some of Mebroot's obfuscated functions were one-to-one with functions implemented in PHIDE2's pullout.c and search.c. Mebroot uses PHIDE2's code coverage algorithm to find all code blocks belonging to the NdisMRegisterMiniport function and its subfunctions and then analyses these blocks to locate the static pointers that represent these two unexported globals. The exact algorithm is represented below.

How Mebroot finds the address of unexported NdisMiniDriverListLock:

1. Finds all globals used by NdisMRegisterMiniport by calling FindSharedGlobals().
2. For every global, enumerates ndis.sys relocations by calling ParseRelocs() and checks whether the relocation target value equals the current global.
3. If match, checks whether the preceding instruction is 'MOV ECX, IMM32'.
4. If true, checks whether the next instruction is 'CALL m32'.
5. If true, checks whether 'm32' equals one of four spinlock API functions[1].
6. If true, current global is NdisMiniDriverListLock.

In our test systems the algorithm found the list lock address from inside the NdisEnumerateInterfaces function as can be shown from the following kernel debugger printouts:

```
kd> u 8c3d+ecx-1
e1c08c3c b96c6b0100      mov     ecx,16B6Ch
e1c08c41 ff15545e0100    call    dword ptr ds:[15E54h]

kd> u 8c3d+ndis-1
NDIS!ndisEnumerateInterfaces+0x172:
f96ffc3c b96cdb6ff9      mov     ecx,offset NDIS!ndisM
iniDriverListLock (f96fdb6c)
f96ffc41 ff1554ce6ff9    call    dword ptr [NDIS!_imp_
KfReleaseSpinLock (f96fce54)]
```

How Mebroot finds the address of unexported NdisMiniDriverList:

1. Acquires NdisMiniDriverListLock spinlock.
2. For every global, enumerates ndis.sys sections and checks whether the global is located within a section that has the following flags enabled:
   a. IMAGE_SCN_MEM_NOT_PAGED
   b. IMAGE_SCN_MEM_READ
   c. IMAGE_SCN_MEM_WRITE
3. If true, reads DWORD from the current global. This is assumed to point to the _NDIS_M_DRIVER_BLOCK structure.
4. Checks pointer validity with MmIsAddressValid and MmIsNonPagedSystemAddressValid.
5. If valid, traverses all _NDIS_M_DRIVER_BLOCK->NextDriver pointers and validates them (step 4) until NULL or invalid pointer is reached.
6. If pointer count > 2, reads DWORD from _NDIS_M_DRIVER_BLOCK->MiniportQueue. This points to the _NDIS_MINIPORT_BLOCK structure.
7. Traverses all _NDIS_MINIPORT_BLOCK->NextMiniport pointers and validates them (step 4) until NULL or invalid pointer is reached.
8. If pointer count > 0, releases the spinlock and the global is NdisMiniDriverList.

These two algorithms are much more system independent than basic signature-based ones. A clear indication that this approach works is that during our tests with different patch and service pack levels we did not encounter a single system crash. It seems that kernel-mode malware has found one more reliable trick to add to its toolbox which allows it almost unlimited access to the non-documented and unexported kernel internals.

BACKDOOR COMMUNICATION AND ENCRYPTION

We tried to study and analyse some infected Mebroot machines in our labs to understand the communication protocol with the C&C

[1] KfAcquireSpinLock, KfReleaseSpinLock, KefAcquireSpinLockAtDpcLevel or KefReleaseSpinLockFromDpcLevel.

server (the 'mothership' server). The Mebroot driver contains some hardcoded hostnames that are used in the code as a first point of contact during network activity. If these hosts are offline, Mebroot is able to generate a pseudo-random hostname using the template '%c%c%c%c%s'. This hostname is based on the current date/month and contains a three-char suffix (%s) which is chosen from a list of known hardcoded strings (e.g. anj, ebf, arm, pra, aym, unj, ulj, uag, esp, kot, onv, edc). This list may change from variant to variant. The final domain suffix is chosen using '.com', '.net' and '.biz' combinations. Earlier variants of Mebroot seem to get the current date using OS functions, but we recently spotted a variant that possibly tries to get the current date by parsing HTTP headers from a GET request to multiple 'google.*' servers. The date prediction algorithm is very similar to the one used by Trojan.Anserin years ago. More information about this algorithm is discussed in [12].

When a generated hostname resolves via DNS to a valid IP address, Mebroot sends an initial encrypted packet to the C&C server using the HTTP protocol. All network traffic is sent through the private TCP/IP stack and therefore is virtually invisible to any application running on the infected host. An example of plaintext and an encrypted packet are shown in Figure 11.

The size of the initial encrypted packet sent out has an encrypted size of 92 bytes (0x5C). The first DWORD in the packet is always the decryption key and the second DWORD gets decrypted to the real size of the cleartext payload. The reply packet has to contain the same key as the first DWORD otherwise the packed will be discarded. The initial packet contains the magic command 'BIP', which is probably used to ping the C&C server. This magic command is followed by other values as explained in the packet structure reconstruction shown in Table 2.

The encryption algorithm used is unknown and protected from immediate reversing by the crazy obfuscation layer present in the rootkit code. However, it is possible to find in the encryption and decryption routines the magic constants of the SHA-1 hashing algorithm. The initial packet shown in Figure 11 is used to communicate to the C&C server the secondary encryption key which the downloaded user-mode payloads can use for further communications with the server. The secondary key is

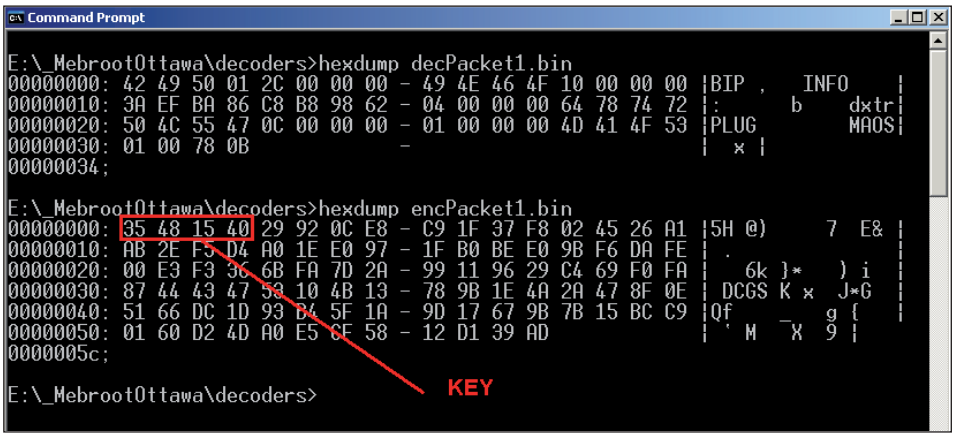

Figure 11: Example of plaintext and encrypted packet sent by Mebroot.

OFFSET	SIZE	NOTE
0x00	DWORD	"BIP\x01"
0x04	DWORD	sizeof(full packet)
0x08	DWORD	"INFO"
0x0C	DWORD	sizeof(Secondary Key)+sizeof(client-id)+4
0x10	8-bytes	Secondary Key for user-mode encryption
0x18	DWORD	sizeof(client-id)
0x1C	DWORD	client-id string (e.g. "dxtr", "grey",)
0x20	DWORD	"PLUG"
0x24	DWORD	sizeof(version-id) + 8
0x28	DWORD	*version-id?* (e.g. 0x00000001)
0x2C	DWORD	"MAOS"
0x30	DWORD	*unknown?*

Table 2: Possible structure of Mebroot initial packet.

usually eight bytes long and generated with a complex hashing routine based on individual characteristics of the hard drive and two magic values 0xBAD1D111 and 0xBAD1D222. The secondary key is used with the old 'Torpig-style' encryption algorithm which is much simpler (xor + base64 encoding). This second algorithm/key combination is responsible for all bank-stealing communications and activities performed later by the downloaded DLL modules.

DLL Encrypted Text:

```
'cEJmlWUXX1TUjnRv/+7ldPVnJ2Uhwqu1Z7t2W7osElZnVXaVBNfe
VAXeVP5vH6Lh1XcnRUEiG9RQ2hHrDs5UcQBiBpUFZs4xxVpxOxp75
XUUt0JxIOJLBVDaIXsq2rVRIKNm1VGSP6AhO5EPfg'
```

DLL Decrypted Text:

```
"ts=0&ip=192.168.146.137:&sport=4203&hport=4234&os=
5.1.2600&cn=Ireland&nid=3AEFBA86C8B89862&bld=mlucky&
ver=204"
```

Table 3: Example of secondary encryption used by Mebroot DLL modules.

During our tests we have seen Mebroot responding to the initial 'BIP' packet with a big encrypted blob of data of about 230 KB. So far, this packet has always contained encrypted images of two DLL modules and instructions to inject them into carefully selected processes running on the infected system. Figure 12 shows the contents of the reply packet after decryption.

From Figure 12 we can easily see what is going on – the infected client is instructed to install two separate user-mode payloads. The first DLL will be injected into 'services.exe' and the second will be injected into multiple processes if they exist on the system. During our tests we did not see any commands other than 'INST' from those servers that we were monitoring. However, based on static code analysis we can conclude with some certainty that Mebroot contains the following backdoor capabilities:

1. Install user-mode DLL into any process or install new version of Mebroot into raw sectors.
2. Uninstall user-mode DLL or uninstall Mebroot from raw sectors.
3. Instruct a trusted process to launch new process by filename.
4. Execute any driver in kernel mode.

What makes Mebroot's backdoor extremely powerful is the fact that the payloads are never stored on disk in cleartext (except for process creation) which means that only memory-based AV scanners are able to detect the downloaded payloads by signature. To understand better how the main payload installation is done we will look in more detail at what will happen when an infected client receives the 'INST' command from the server.

The installation routine will first enumerate all files under the 'system32' directory and select a random file from it. It removes unnecessary headers from the decrypted packet and encrypts it again using the same algorithm that is used for the network traffic. The new encrypted payload is stored in a new file under the 'system32' directory whose name equals the previously selected benign file with the file extension changed to a random one. Also, the file properties, such as creation and modification time, are changed to match the benign file. This behaviour is illustrated in Figure 13 where the payload is stored in a file named 'qdv.jsx' and both its encrypted and cleartext contents are shown.

Once the payload is successfully stored on disk the first stage of the installation process is completed. After this point the payload can be read from the file, decrypted in memory and reloaded after reboot even if the C&C server goes offline.

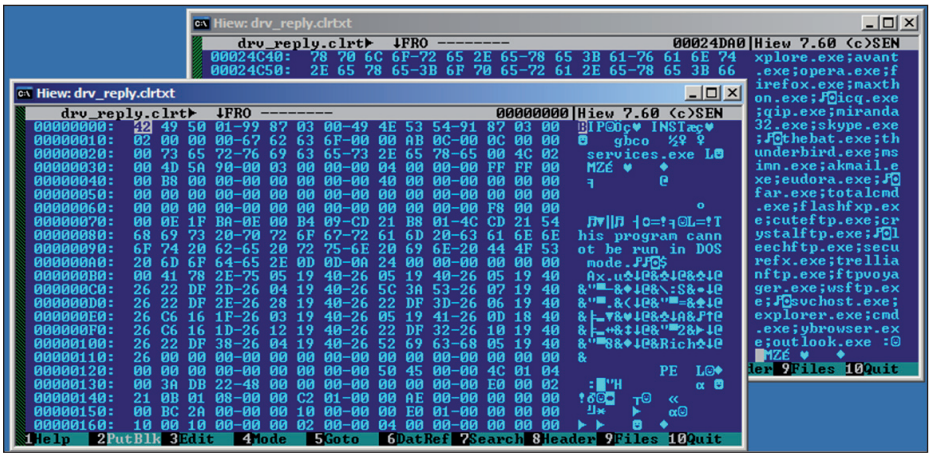

Figure 12: Reply packet from Mebroot C&C server after decryption.

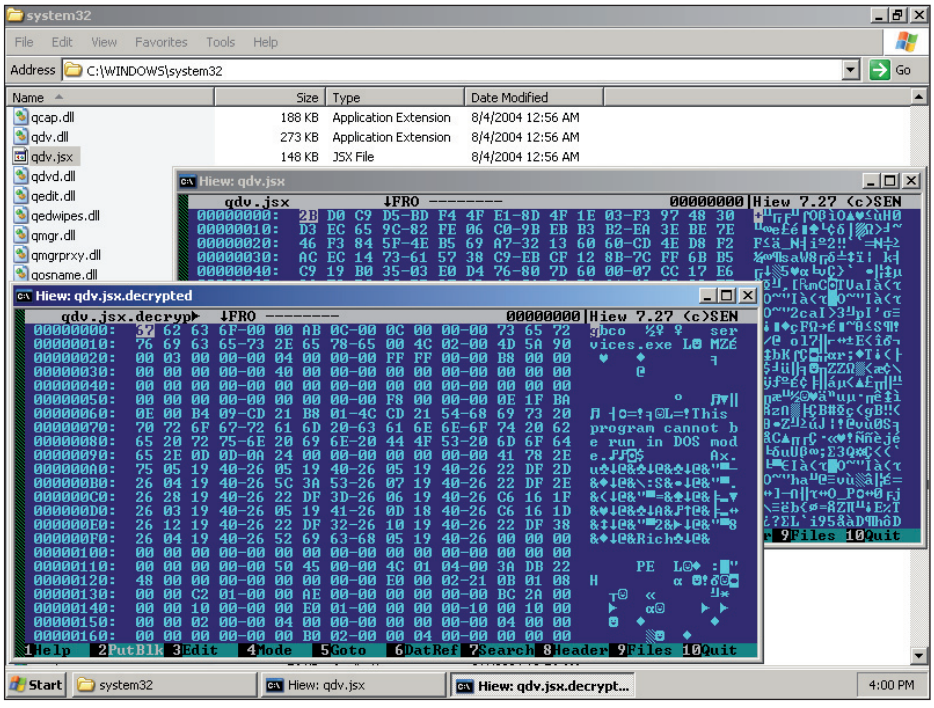

Figure 13: Downloaded payload is stored on disk in an encrypted file.

The next phase is to execute the payload. As we have mentioned the payload has so far contained two DLLs that will be injected into selected processes. In fact we can see from Figure 13 that the stored payload contains the target process name ('services.exe') and the size of the PE file right before the 'MZ' header. The second DLL, its size and list of target processes will follow right after this block. To load the DLL into the target process(es) the code executes the following steps:

1. Finds target process object and changes into its context by executing KeStackAttachProcess.
2. Calls NtAllocateVirtualMemory to allocate RWX memory from user mode for custom DLL loader module and the actual DLL.
3. Prepares the loader module and the DLL for execution by copying them into the allocated buffers and by relocating them to their new base addresses.
4. Finds an alertable thread and queues a user-mode APC that will execute the entry point of the loader module.
5. Waits for an event from the loader module to indicate that it has finished its job.

The loader module, which is a DLL itself, will first resolve its own imports and then destroy its PE headers to maintain stealth. Next it

will resolve the imports for the injected DLL and call its entry point. Finally it calls SetEvent to signal to the driver that the payload has been loaded.

The interesting finding is that the Mebroot driver provides 21 (0x15) kernel-mode routines that the injected DLLs can utilize through a shared memory buffer and signalling event. We also found another reference to the routine array from code that uses an array of 21 three-letter strings as the selector. This routine is part of the backdoor's kernel-mode payload handler and would indicate that the server can instruct the client to execute any of these routines. The following are some example strings:

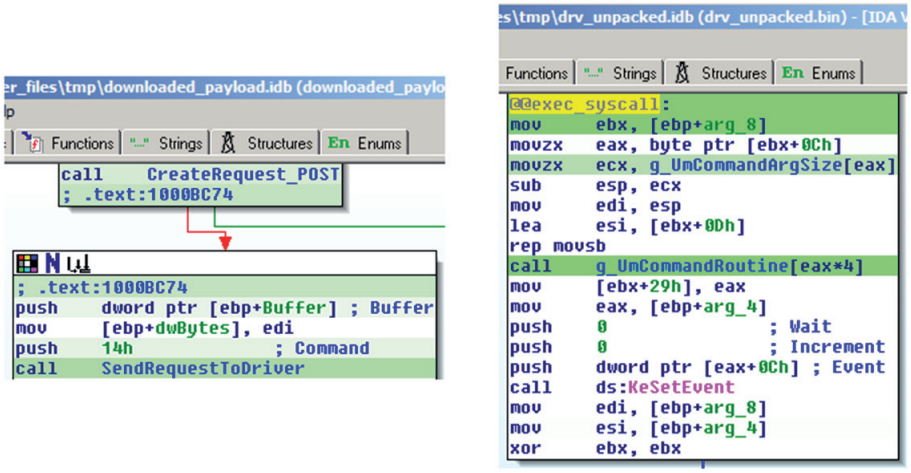

Figure 14: Downloaded DLL requests the driver to send the POST request on its behalf.

- 'the', 'mat', 'rix', 'has', 'you', 'neo'

The purpose of each 'system call' routine is still a little unclear but we have been able to confirm that several of them are related to the driver's private TCP/IP stack and at least the first DLL uses it to communicate with the C&C server and therefore can effectively bypass any personal firewall or network-monitoring software running on the infected host. One example of the private system call interface is illustrated in Figure 14.

Based on the analysis we have done so far it is clear to us that Mebroot's backdoor capabilities are powerful – the most powerful we have seen so far. It is capable of pushing any DLL into infected clients and the DLL can be modified to utilize the private system call interface with minimal changes, thus allowing it to take full advantage of the services provided by the Mebroot driver. In addition to this, the backdoor can upload and execute any kernel-mode module it wants on the client.

Now that we have been able to get a slightly bigger picture of this monstrous beast we have started to wonder what this mysterious 'MAOS' string means, which is clearly some kind of signature from the authors. Based on what we have seen so far we are starting to think that it could mean 'Malware Operating System' – an evil operating system running inside the real one!

The positive side is that Mebroot seems to have an uninstallation routine that the C&C server can instruct the infected clients to execute. Since the key exchange protocol between the client and the server is weak we have been able to break the encryption and could theoretically hijack the botnet and instruct every infected client to clean themselves. However, since this approach might have some legal implications we have so far used this weakness just to decrypt the C&C traffic from network captures to understand Mebroot's inner workings in more detail.

CONCLUSIONS

Mebroot is the most advanced and stealthiest malware we have analysed so far. It operates in the lowest levels of the operating system, uses many undocumented tricks and relies heavily on unexported functions and global variables. Still we did not encounter a single blue screen with the latest samples that were distributed after February 2008. This is a clear sign of the level of professionalism the malware authors are practising today. It is also evident that the authors of Mebroot are closely following the research done by individuals often presenting their findings at Black Hat conferences or on rootkit.com. Mebroot's MBR code is almost identical to Bootroot's, while the firewall-bypassing code closely follows the most advanced ideas presented by Tereshkin at Black Hat USA 2006 [13]. In addition, after we successfully unobfuscated some of the code used to perform the code pullout it became clear that some of the functions were one-to-one with functions that are part of the PHIDE2 source code. Maybe the next malware from Mebroot's author will be utilizing virtualization to make it even more difficult to detect and remove – at least proof of concept source code for this is already available [14].

REFERENCES

[1] Master Boot Record timeline. http://isc.sans.org/diary.html?storyid=3820.

[2] GMER team. Stealth MBR rootkit (2 Jan 2008). http://www2.gmer.net/mbr/.

[3] The Flow of MBR Rootkit Trojan Resumes. http://www.symantec.com/enterprise/security_response/weblog/2008/02/the_flow_of_mbr_rootkit_trojan.html.

[4] Hoglund, G.; McGraw, G. Exploiting Software. 2004. p.429.

[5] eEye BootRoot. http://research.eeye.com/html/tools/RT20060801-7.html.

[6] BOOT KIT: Custom boot sector based Windows 2000/XP/Vista subversion. http://www.nvlabs.in/?q=node/11 .

[7] INFO: Direct Drive Access Under Win32, Microsoft. http://support.microsoft.com/kb/q100027.

[8] Rutkowska, J. Subverting Vista kernel for fun and profit. 2006.

http://www.invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt.

[9] Kapoor, A.; Mathur, R. Strike me down, and I shall become more powerful! Virus Bulletin. June 2008. pp.8–10. http://www.virusbtn.com/virusbulletin/archive/2008/06/vb200806-strike-me-down.

[10] Kasslin, K.; Florio, E. Spam from the kernel. Virus Bulletin, November 2007, pp.5–9. http://www.virusbtn.com/virusbulletin/archive/2007/11/vb200711-srizbi.

[11] Phide2. http://rootkit.com/vault/90210/phide2.zip.

[12] Ligh, M.H. MBR rootkit domain name prediction algorithm. http://mnin.blogspot.com/2008/01/mbr-rootkit-domain-name-prediction.html.

[13] Tereshkin, A. Rootkits: attacking personal firewalls. Black Hat USA 2006. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf.

[14] Blue Pill Project, http://bluepillproject.org/.